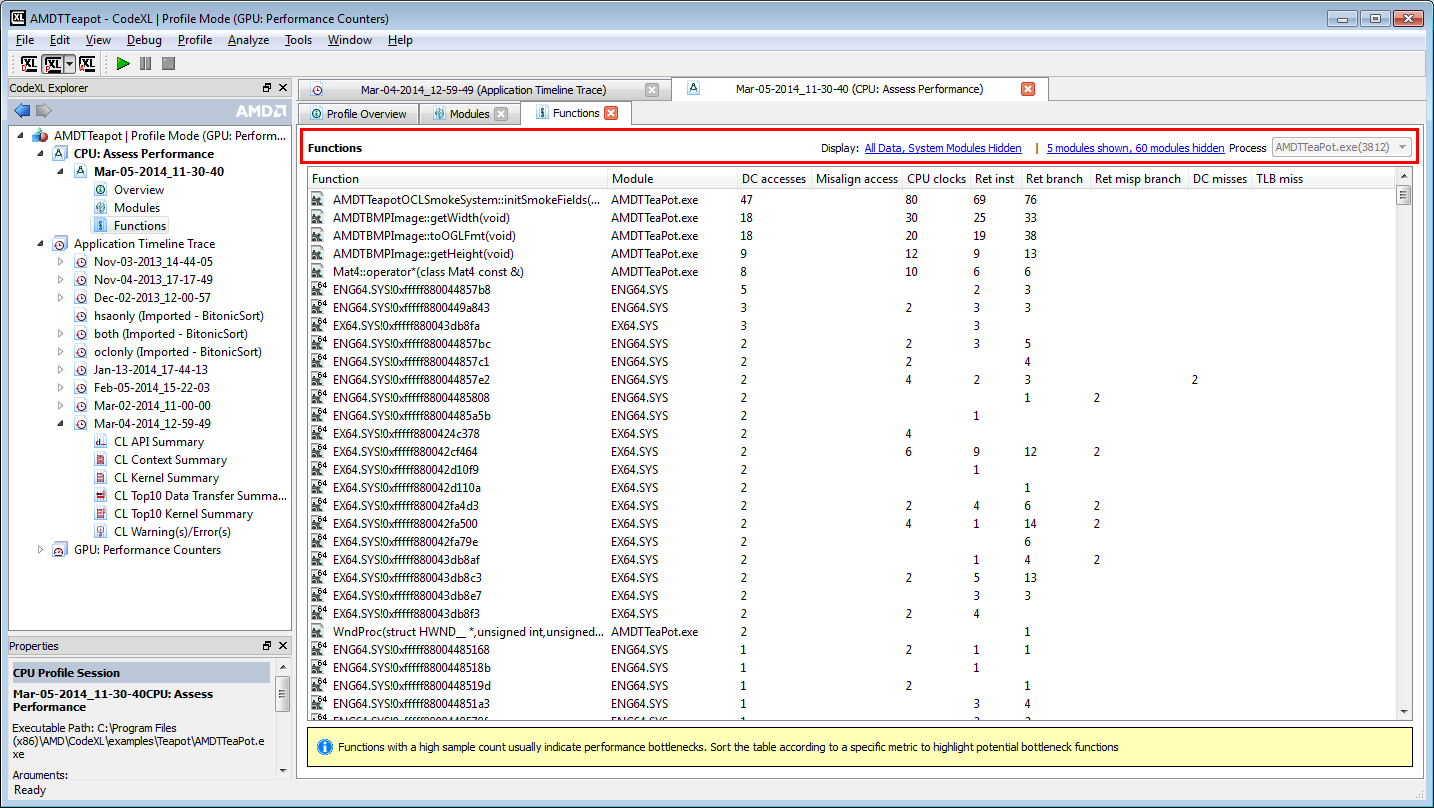
# acToolbar

All the CPU profile inner views contain toolbars. These toolbars are located in the top of the windows, and used to give the user set of tools for configuration of the current view.

For example, the following screenshot is a display of a functions view. The top toolbar is marked in red rectangle. We used to add these controls to the view by using simple layout (QHBoxLayout), but this didn’t work. Qt didn’t know how to handle resize, and controls were overlapping one another.



At first step, we switched from layout to QToolbar. The problem was that when using a QToolbar for inner windows, Qt didn’t have the simple ability for resize it had for toolbars used in main windows. When the window was resized, and the toolbar did not have enough width to paint it’s widgets, Qt would paint >> like in the below screenshot. But these arrows were not clickable.



So, for narrow windows our toolbar would lose some of its widgets.

In Qt documentation we saw that when using a toolbar in windows that are not QMainWindow children, we should implement the toolbar differently. The exact quote from the documentation is:

“When a QToolBar is not a child of a QMainWindow, it loses the ability to populate the extension pop up with widgets added to the toolbar using addWidget(). Please use widget actions created by inheriting QWidgetAction and implementing QWidgetAction::createWidget() instead.”

In order to resolve this issue, we implemented acWidgetAction, which inherits QWidgetAction. When we use a toolbar in windows which are not children of QMainWindow, we use these actions to add widgets to the toolbar, and not the naïve “addWidget” function.

**acWidgetAction** isthe class handling the creation of the widget, and Qt calls it whenever the update of the geometry of the window is needed.

Currently, acWidgetAction supports the following types of widgets: QWidget / QLabel / QComboBox.

If in the future we will need to add another type of widget to the inner toolbars, we should follow what we do for labels and combo boxes and add functions handling it.

There are few ways to add a widget to the toolbar:

* Call AddWidget. This function is generic and get a structure of type acToolbarActionData, which defines the attributes for the added widget.
* Call AddLabel / AddComboBox.

If the widgets should be updated online, the way to update it is to save the action returned from the above functions, and update its properties using the access functions of acWidgetAction.

For example: if a combo box list is updated online, do the following:

* Define a member for the combo box action:

acWidgetAction\* m\_pComboBoxAction;

* Create the combo box action:

m\_pComboBoxAction = pToolbar->AddComboBox(listOfStrings, SIGNAL(currentChanged), this, SLOT(OnComboChanged)));

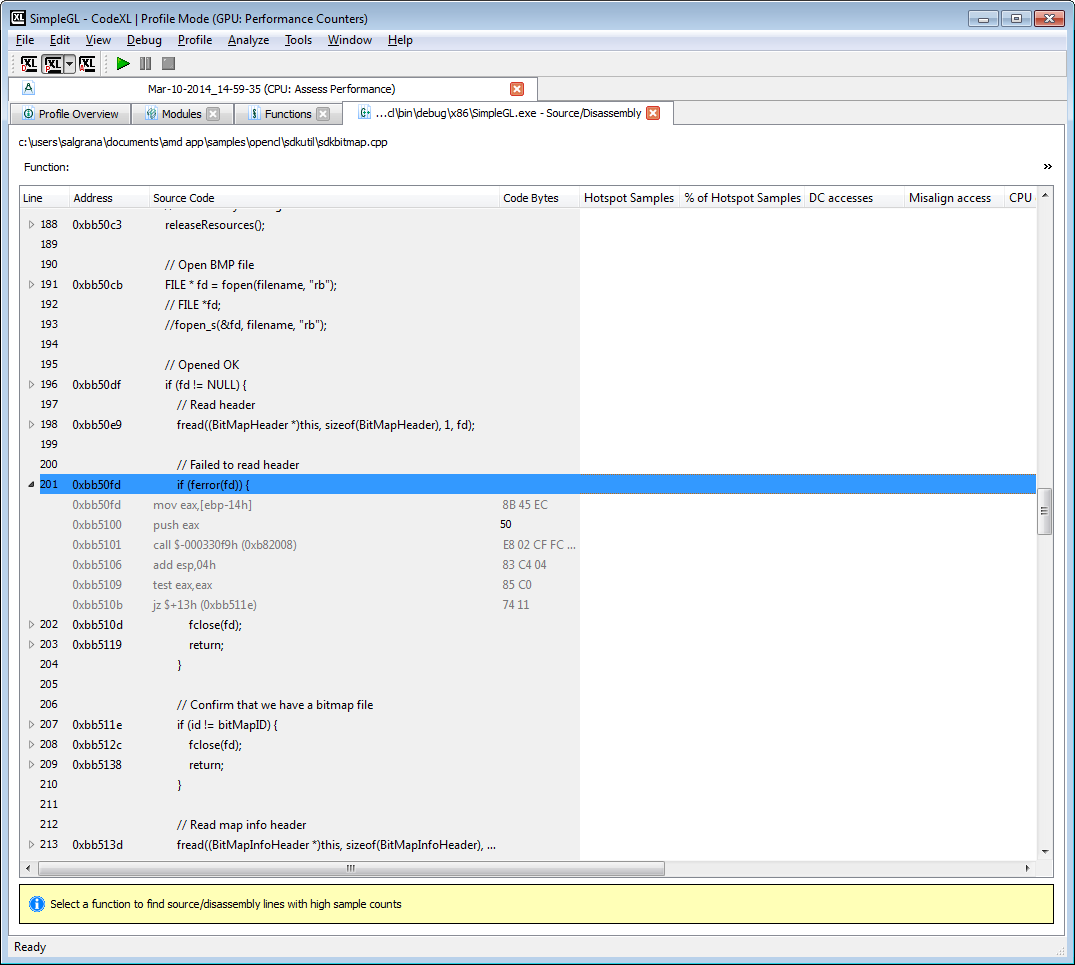
* When needs to update the list of strings for instance:

m\_pComboBoxAction->UpdateStringList(newListOfStrings);

# Source Code View Tree

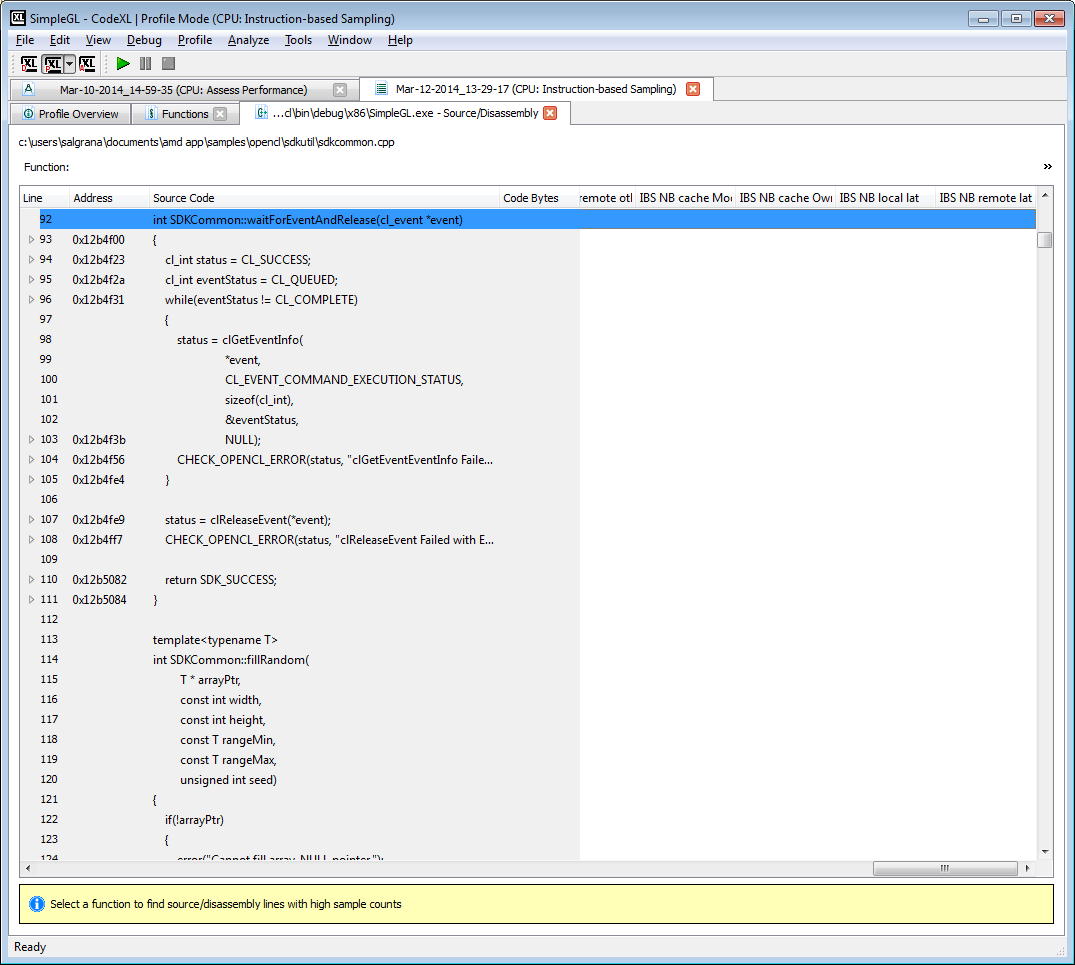
The source code view tree is used to display the following columns:

1. Line number, Address, Source Code, Code Bytes – 4 columns that depend only on the source code, and not on the profile session results.
2. Hotspot samples, % of Hotspot samples – is displaying the current hotspot indicator samples count and percentage.
3. Columns that display all the counters values.



It is important for us that while the user scroll horizontally to the right, he will be able to the 4 left columns fixed to the left. If we have many counters, this functionality becomes more important.

So we had to implement a tree view with “frozen columns”. This tree receives number of frozen columns as input, and makes sure that no matter how right the scroll will go, this number of columns will stay fixed to the left side of the view. In the following screenshot, for example, there are many counters displayed. See the scroll size; it is small since we have large scroll range. The scroll is taken to its rightmost position, and still the gray columns are displayed, fixed to the left side of the view.



The object which in which this tree is implemented at is **acFrozenColumnTreeView**.

This tree is a child of QTreeView (Qt tree display), and has a member: **m\_pFrozenTreeView.** Both trees are using the same model (displaying the same data base). The main tree (the class), is displaying columns 0 - m\_frozenColumn (which can be configured from the user of the tree), and m\_pFrozenTreeView is displaying m\_frozenColumn – last column.

Vertical scrolls for both tree are aligned (see OnVerticalScrollPositionChanged. It is making sure that once a scroll is changed in one tree, it is changed in the other as well.

In order to make sure that the horizontal scroll is leaving the frozen columns displayed when they should be hidden, the function **moveCursor** is overridden.

This is the function documentation description:

“Returns a QModelIndex object pointing to the next object in the view, based on the given cursorAction and keyboard modifiers specified by modifiers.”

We re-implement the action **MoveLeft**. We test the geometry of the item that should be moved and override the scroll bar value accordingly.

**Notice**: since this is a tree view that is constructed from 2 trees, we should not connect to QTreeView signals. If we need to connect to such signal, we should implement a new signal in **acFrozenColumnTreeView**, connect to the original 2 trees signals, and fire the new created signal instead of the original one.

For example, follow what we do in **OnItemDoubleClick**.